FUNKTIONAALISTEN OMINAISUUKSIEN HYÖDYT

Scala-ohjelmointikielessä

Tuomo Häkkinen

Opinnäytetyö
Toukokuu 2012

Ohjelmistotekniikan koulutusohjelma
Tekniikan ja liikenteen ala

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖN
KUVAILULEHTI

Tekijä(t) HÄKKINEN, Tuomo	Julkaisun laji Opinnäytetyö	Päivämäärä 16.5.2012
	Sivumäärä 49	Julkaisun kieli Suomi
	Luottamuksellisuus ( ) saakka	Verkkojulkaisulupa myönnetty (X)

Työn nimi
FUNKTIONAALISTEN OMINAISUUKSIEN HYÖDYT. SCALA-OHJELMOINTIKIELESSÄ.

Koulutusohjelma
Ohjelmistotekniikka

Työn ohjaaja(t)
PELTOMÄKI, Juha

Toimeksiantaja(t)
Viklo Oy

Tiivistelmä

Opinnäytetyössä tutkittiin funktionaalisten ominaisuuksien hyötyjä Scala-ohjelmointikielessä. Tuloksien voidaan kuitenkin olevan yleisluontoisia kaikille funktionaalisille ohjelmointikielille.
Työssä käsiteltiin erilaiset ohjelmistoparadigmat, funktionaaliset ominaisuudet sekä niiden käytöstä aiheutuvat hyödyt.

Funktionaaliset ominaisuudet ja niiden hyödyt esiteltiin Scala-ohjelmointikielellä kirjoitetuilla lyhyillä esimerkeillä. Joissakin esimerkeissä käytettiin myös Java-ohjelmointikieltä, kun vertailtiin imperatiivisen sekä funktionaalisen eli deklaritiivisen ohjelmoinnin eroja. Joissakin esimerkeissä imperatiivisesti kirjoitetty ohjelmointikoodi muutettiin funktionaalisemmaksi, jotta ymmärrettäisiin funktionaalinen ajatusmaailma.

Tutkimuksen tulokset kannustavat käyttämään funktionaalisia ominaisuuksia. Tulokset vahvistavat myös Viklo Oy:n käsityksiä funktionaalisten ominaisuuksien hyödyistä, ja rohkaisevat käyttämään tulevaisuudessakin Scala-ohjelmointikieltä ohjelmistoprojekteissaan. Tutkimuksesta on hyötyä myös ohjelmoijille, jotka ovat vasta tutustumassa funktionaaliseen ohjelmointiin tai Scala-ohjelmointikieleen.

Avainsanat (asiasanat)

Funktionaalinen ohjelmointi, olio-ohjelmointi, Scala, funktionaalinen ominaisuus

Muut tiedot

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

DESCRIPTION

Author(s) HÄKKINEN, Tuomo	Type of publication Bachelor´s Thesis	Date 16052012
	Pages 49	Language Finnish
	Confidential ( ) Until	Permission for web publication (X)
Title BENEFITS OF FUNCTIONAL FEATURES. IN SCALA-PROGRAMMING LANGUAGE.		
Degree Programme Software Engineering		
Tutor(s) PELTOMÄKI, Juha		
Assigned by Viklo Oy		

Abstract

The purpose of the bachelor´s thesis was to find out the benefits of functional features in Scala-programming language. The results still can be said to be general in all functional programming languages. The thesis deals with different programming paradigms, functional features and the benefits caused from the use of functional features.

Functional features and their benefits were presented by small examples programmed in Scala-programming language. Some examples were written in Java-programming language when imperative and declarative programming differences were compared. In some examples, imperatively written program code was changed into more functional in order to understand the functional mindset.

Results of this study encourage the use of functional features. The results also confirm Viklo Oy´s belief of the benefits of functional features and will encourage the use of Scala-programming language in their software projects in the future. The study is also useful for programmers who are not familiar with functional programming or Scala-programming language.

Keywords
Functional programming, object-oriented programming, Scala, functional feature

Miscellaneous

SISÄLTÖ

SANASTO 3
1 TYÖN LÄHTÖKOHDAT 5
1.1 Toimeksiantaja 5
1.2 Tavoite 5
2 OHJELMOINTIPARADIGMAT 7
2.1 Ohjelmointikielten historia 7
2.2 Proseduraalinen ohjelmointi 9
2.3 Logiikkapohjainen ohjelmointi 9
2.4 Oliopohjainen ohjelmointi 10
2.5 Funktionaalinen ohjelmointi 13
2.6 Scala 16
3 FUNKTIONAALISET OMINAISUUDET 18
3.1 Sivuvaikutuksettomuus 18
3.2 Rekursio 22
3.2 Muuttumattomat tietorakenteet 25
3.2.1 Lista 25
3.2.2 Tuple 26
3.2.3 Map 27
3.3 Ensimmäisen luokan funktiot 28
3.3.1 Literaalifunktio ja sen asettaminen muuttujaan 28
3.3.2 Funktio funktion argumenttina 31
3.3.3 Funktion palautusarvona funktio 31
3.3.4 Sisäkkäiset funktiot 32
3.4 Osittain sovellettu funktio 33
3.5 Closure 35
3.6 Currying 35
3.7 Pattern matching 37
3.8 Laiskat muuttujat 38
4 FUNKTIONAALISTEN OMINAISUUKSIEN HYÖDYT 40

4.1 Lyhyempää ja tuottavampaa koodia 40
4.2 Turvallisuus 41
4.3 Helppo testattavuus 42
4.4 Uudelleenkäytettävyys ja laajennettavuus 42
4.5 Selkeys 44
4.6 Rinnakkaisuus 44
5 POHDINTA 45

SANASTO

Aliohjelma	Metodi, funktio tai proseduuri.
Argumentti	Funktiolle välitettävä tieto.
Funktio	Funktionaalisen ohjelman toiminto.
Hajautustaulu	Avaimia arvoihin yhdistävä tietorakenne.
Java	Oliopohjainen ohjelmointikieli.
JVM	Java Virtual Machine, käyttöympäristöön asennettu ajonaikainen virtuaalikone, jossa käännettyä tavukoodia ajetaan.
Induktio	Matemaattinen todistusmenetelmä.
Kääntäjä	Ohjelma, joka kääntää lähdekoodin bittikoodiksi (konekieliseksi).
Metodi	Oliopohjaisen ohjelman toiminto.
Moduuli	Ohjelman itsenäinen osa, jolla on jokin oma toiminnallinen tehtävä.
Ohjelmointiparadigma	Ohjelmointikielen taustalla oleva tapa ajatella ja mallintaa ongelmien ratkaisuja.
Parametri	Metodille välitettävä tieto.

Proseduuri	Proseduraalisen ohjelman toiminto.
Scala	Ohjelmointikieli, joka yhdistää olio- ja funktionaalisen ohjelmoinnin.
Silppujoukko	Tietorakenne, jonka elementeille täytyy olla laskettavissa hajautuskoodi, joka on yksilöllinen tunniste elementille.
Sovelluskehys	Ohjelmistotuote, joka toimii apuvälineenä uusien ohjelmistotuotteiden nopeampaan valmistukseen.
Squeryl	DSL (Domain-specific language) tietokantaobjektien käsittelyyn Scala-ohjelmointikielellä.
Säie	Ohjelman itsenäisesti suoritettava osa moniajojärjestelmissä.
Vaadin	Sovelluskehys käyttöliittymiä varten.
Vuonohjauslause	Ohjelman komento, joka siirtää ohjelman suorituksen jatkumaan muualta kuin järjestyksessä seuraavasta komennosta.

1 TYÖN LÄHTÖKOHDAT

1.1 Toimeksiantaja

Opinnäytetyön toimeksiantajana toimi Viklo Oy. Viklo on Jyväskylässä joulukuussa 2010 perustettu ohjelmistoyritys. Yrityksen ydinosaamista ovat johdon raportointijärjestelmät, tietovarastointi sekä toiminnanohjausjärjestelmien ja sovelluskehityksen konsultointi. Viklo panostaa uusimpiin teknologioihin saavuttaakseen maksimaalisen tehokkuuden projekteissaan. Käytettyjä teknologioita sovelluskehityksessä ovat mm. Java, Java EE, Scala, Vaadin sekä Squeryl. Viklon suurimpia asiakkaita ovat Metso Paper sekä Komas. Yrityksessä työskentelee yhteensä seitsemän työntekijää erilaisissa työtehtävissä. Sovelluskehityksestä vastaa kolme henkilöä. (Viklo Oy 2012.)

1.2 Tavoite

Tavoitteena oli selvittää funktionaalisten ominaisuuksien hyviä puolia. Funktionaalisena kielenä käytettiin opinnäytetyössä Scalaa. Haskell-ohjelmointikieleen tutustuttiin myös hieman opinnäytetyöprosessin aikana. Scalan valinta funktionaaliseksi kieleksi oli helppoa, sillä sitä myös käytettiin yrityksessä ohjelmointikielenä. Ei-funktionaaliseksi vertailukieleksi valittiin Java, jonka opinnäytetyön tekijä tunsi vahvimmaksi ohjelmointikielekseen. Kaikki opinnäytetyössä esitetty koodi on joko Scalaa tai Javaa, jotta lukijan on helpompi ymmärtää lukemaansa eikä siirrytä useasta ohjelmointikielestä toiseen.

Yrityksellä oli tarve saada esiin funktionaalisten ominaisuuksien hyötyjä siirryttäessä yhä enemmän Scala-ohjelmointikielen käyttöön. Myös tulevien projektien ohjelmointikielten valinta voisi helpottua tuntemalla funktionaaliset hyödyt. Täten voitaisiin valita paras vaihtoehto ohjelmointikielistä vastaamaan projektin vaatimuksia tai jotakin sovelluksen osa-aluetta.

Pohjatietona opinnäytetyöhön toimi olio-ohjelmoinnista kertynyt kokemus usean

vuoden varrelta. Funktionaaliset kielet olivat opinnäytetyön tekijälle lähes täysin uusia. Scala-ohjelmointikieltä ja sen funktionaalisia ominaisuuksia kuitenkin oppi päivittäin työelämässä.

2 OHJELMOINTIPARADIGMAT

2.1 Ohjelmointikielten historia

Ohjelmointikielet voidaan jakaa proseduraalisiin, oliopohjaisiin, funktionaalisiin sekä logiikkapohjaisiin ohjelmointikieliin. Ne ovat kaikki vielä varsin nuoria. Ensimmäistä toimivaa ohjelmointikieltä alettiin kehittää vuonna 1943. Se oli konekohtainen, eli kieltä voitiin käyttää sellaisenaan vain yhdessä koneessa. Kieli sai nimekseen ENIAC. Ohjelmointi oli vielä tähän aikaan hyvin vaikeaa. (List of programming languages 2012.)

1950- ja 60-luvuilla ohjelmointikielet kehittyivät paljon. Ne eivät enää olleet konekohtaisia. Reaalimaailman ongelmien mallintamisen tarve johti olioajattelun kehittymiseen. Erilaisia ohjelmointikieliä oli kehitetty jo kymmeniä 60-luvun loppuun mennessä. Osa näistä kielistä on vieläkin käytössä, kuten 50-luvulla kehitetyt FORTRAN(1957), LISP(1958) sekä COBOL(1959) sekä 60-luvulla kehitetty Simula. LISP oli ensimmäinen funtionaalinen ohjelmointikieli ja Simula ensimmäinen olio-ohjelmointia tukeva kieli. (List of programming languages 2012.)

1970-luvulla ohjelmointikieliä kehitettiin vielä paljon enemmän edellisiin vuosikymmeniin nähden. Myös ensimmäinen puhdas olio-ohjelmointikieli Smalltalk kehitettiin tällöin. Se oli kuitenkin vielä hyvin kehittymätön olio-ominaisuuksiltaan. Proseduraalinen C-kieli luotiin vuonna 1972, joka oli yksinkertainen, tehokas sekä joustava. Se kehitettiin alun perin järjestelmäohjelmointiin, mutta sitä on käytetty myöhemmin myös paljon sovellusohjelmointikielenä. C-kielestä kehittyi yksi suosituimmista kielistä, jota käytetään yhä edelleen. Samana vuonna kehitettiin myös ensimmäinen logiikkapohjainen ohjelmointikieli nimeltään Prologic. (List of programming languages 2012.) 1970-luvun tärkeimmät kehitetyt kielet olivat C, Pascal sekä SQL, joka oli aluksi vain kyselykieli. Siihen on myöhemmin kuitenkin lisätty ohjelmoinnin rakenteita. Suurin osa ohjelmointikielten ajatusmalleista, jotka ovat vieläkin käytössä, keksittiin 1970-luvulla. Niitä ovat esimerkiksi toistorakenteet, aliohjelmat, muuttujien staatti-

nen tyypitys sekä muuttujien vaikutusalueen rajoittaminen.

1980-luvulla ohjelmointikielissä otettiin käyttöön moduulit suuria järjestelmiä varten. Myös olio-ohjelmointi lisäsi suosiotaan C++ ohjelmointikielen sekä graafisten käyttöliittymien lisääntymisen myötä, johon olio-ohjelmointi on omiaan. C++ perustui C-kieleen, johon oli lisätty oliopohjaisia ominaisuuksia. 80-luvulla Yhdysvallat standardisoi proseduraalisen Ada-ohjelmointikielen puolustusvoimia varten. (List of programming languages 2012.) 80-luvun merkittävimpiä kieliä olivat C++, Ada sekä Perl, jotka ovat edelleenkin vahvasti käytössä.

1980-luvulla tuli myös ajatus tekoälyjärjestelmistä, jotka korvaisivat ohjelmoinnin. Tähän ajatukseen liittyviä ohjelmointikieliä ovat funktionaaliset sekä logiikkapohjaiset kielet. Tekoälyjärjestelmissä tavoitteena on luoda tietokone, joka osaisi itse “ajatella” ilman, että tietokone tarvitsisi ihmisen luomaa koodia. Nykypäivänä on olemassa tällaisia järjestelmiä, joita käytetään lääketieteellisessä diagnostiikassa ja erilaisissa päättelytehtävissä. (Tietotekniikan perusteet/1. 2004. Etäopetusmateriaali. Pohjois-Karjalan Aikuisopisto.)

1990-luvulla internetin suosio suorastaan räjähti, millä oli myös suuri vaikutus ohjelmointikieliin. Java-ohjelmointikieltä alettiin kehittää vuonna 1991. Se sai suuren suosion ollessaan täysin oliopohjainen kieli. Internetin myötä myös useat skriptikielet saivat suosiota Perlin ollessa suosituin palvelinten ohjelmoinnissa. Funktionaalinen ohjelmointi lisäsi hiukan suosiotaan, koska se oli tuottavaa. 90-luvulla kehitettiin paljon muitakin suosittuja ohjelmointikieliä kuten Haskell, Python, Java, Ruby, PHP sekä Visual Basic. (Programming language history 2012.)

2000-luvulla internetin suosio on edelleen kasvanut paljon. Nykyisestä webistä käytetään usein termiä web 2.0, jolla viitataan toiminnallisempiin www-pohjaisiin sovelluksiin. Ohjelmointi onkin siirtynyt yhä enemmän palvelinohjelmointiin. Tämän takia ohjelmointikielissä on keskitytty enemmän turvallisuuteen ja luotettavuuteen. Ohjelmoinnissa käytetään nykyisin myös paljon kolmansien osapuolten komponentteja ja sovelluskehyksiä. Vuosikymmenenen alussa kehitettiin täysin oliopohjainen ohjel-

mointikieli C#, joka on kerännyt paljon suosiota Javan ohella. Myös tässä opinnäytetyössä käytetty Scala-ohjelmointikieli kehitettiin 2000-luvun alkupuolella. Erilaisia ohjelmointikieliä on nykyisin olemassa useita satoja. (List of programming languages 2012.) Tässä opinnäytetyössä keskitytään ainoastaan funktionaaliseen sekä oliopohjaiseen ohjelmointiin.

2.2 Proseduraalinen ohjelmointi

Proseduraalinen ohjelmointi on imperatiivista, eli komennot suoritetaan järjestyksessä ensimmäisestä viimeiseen apunaan ehto- ja toistolauseet. Vuonohjauslauseet voivat kuitenkin siirtää ohjelman jatkumisen jostakin toisesta kohdasta. Proseduraalisessa ohjelmoinnissa pyritään jäsentelemään ohjelman suoritusta jakamalla se aliohjelmiin eli proseduureihin. Yksi proseduuri ratkaisee yhden ongelman.

Pääohjelman voidaan sanoa olevan ohjelman sisällysluettelo, joka kutsuu proseduureja eli aliohjelmia, jotka voivat kutsua muita proseduureja. Suuret ohjelmat voidaan jakaa moduuleihin, jotka sisältävät useita proseduureja. Yksi moduuli ratkaisee yhden suuremman ongelman.

Proseduraalinen ohjelmointi on varhaisin paradigma, ja sen suosio on tällä hetkellä edelleen noin 36 prosenttia. Proseduraalinen ohjelmointi on hyvin tärkeää systeemiohjelmoinnissa. Suosituimmat täysin proseduraaliset kielet ovat C, Perl sekä Pascal. (TIOBE Programming Community Index for March 2012. 2012.)

2.3 Logiikkapohjainen ohjelmointi

Logiikkapohjaisessa ohjelmoinnissa kuvataan ratkaisut faktoina ja faktojen välisinä suhteina. Ohjelmat muodostuvat käskyjen sijaan säännöistä ja kyselyistä. Esimerkiksi “Kalle on Matin isä” voitaisiin kuvata isä(Kalle, Matti). Nyt voitaisiin esittää kysely, joka tutkisi, onko Kalle Matin isä.

Logiikkapohjaisia kieliä käytetään lähinnä tekoälytutkimuksen piirissä, jossa se on

syntynytkin (Ohjelmointikielten kehityshistoriaa 2012.). Logiikkapohjaisten ohjelmointikielten suosio on tällä hetkellä vain noin kaksi prosenttia (TIOBE Programming Community Index for March 2012 2012.). Logiikkakielet ovatkin vaikeita vaikka niillä usein saadaan hyviä tuloksia aikaan (Peltomäki & Silander 2003, 117).

2.4 Oliopohjainen ohjelmointi

Oliopohjaisessa ohjelmoinnissa ratkaisut esitetään olioiden yhteistoimintana. Ohjelman rakenne koostuu luokista, jotka mallintavat käsiteltyä tietoa. Luokista luodaan olioita. Luokat sisältävät ominaisuuksia sekä metodeja, jotka määrittelevät kuinka luokan olioiden ominaisuuksia käsitellään. Olio-ohjelmoinnissa pääohjelmaa käytetään yleensä lähinnä ohjelman käynnistymiseen. Se ei siis kontrolloi ohjelman kokonaisuutta, vaan kontrolli etenee luokkien metodien kutsuen toisiaan.

Olio-ohjelmoinnin edut

- Perintä eli luokat voivat periä toisia luokkia, jolloin peritty luokka voi käyttää perittävän luokan metodeja ja ominaisuuksia. Moniperinnällä tarkoitetaan luokan periytymistä useasta luokasta. Java-ohjelmointikielessä moniperintä ei ole mahdollista, eli luokka voi periä ainoastaan yhden luokan, mutta esimerkiksi C++-kielessä voidaan periä useita luokkia.

- Luokkakirjastot, jotka ovat kirjastoja, joihin on määritelty tiettyyn toimintaan kuuluvia perustoimintoja. Esimerkiksi luokkakirjasto “Math” sisältää yleisiä matemaattisia toimintoja. Luokkakirjastojen käyttö helpottaa sovellusten rakentamista.

- Tiedon kapselointi eli itse tieto ja sen toiminnallisuus on kootusti yhdessä oliossa. Olion käyttäjän ei tarvitse käyttövaiheessa tietää olion sisäistä toimintaa, vaan riittää, kun tietää, kuinka oliota käytetään ja kuinka se käyttäytyy. Olion luokka voi sisältää yksityisiä muuttujia tai funktioita, joita itse olio voi ainoastaan käyttää.

- Tietosisältö on usein helppoa mallintaa reaalimaailman käsitteillä.

Olio-ohjelmointi tuottaa oikein käytettynä helposti **uudelleenkäytettävää, hallittavaa sekä ymmärrettävää koodia**.

Olio-ohjelmoinnin haitat

- Olio-ominaisuuksien systemaattinen käyttö voi olla raskasta ohjelman suorituksen kannalta.

- Virheet monistuvat kaikkia oliota käyttäviin ohjelman osiin.

- Tietosisällön rakenteen ollessa yksinkertainen proseduraalinen kieli on usein luontevampaa.

- Ohjelmointiparadigman opettelu vie aikaa. Olio-ohjelmointia täytyy osata hyödyntää oikealla tavalla.

Olio-ohjelmoinnin suosio

Suosituimmat oliopohjaiset ohjelmointikielet ovat Java, C# sekä C++. Niitä suositaan, koska ne tarjoavat kehittyneitä välineitä ongelmien ratkaisemiseen. Olio-ohjelmointikielet ovat nykyisin käytetyimpiä ja niiden osuus on noin 58 prosenttia kaikista ohjelmointikielistä. (TIOBE Programming Community Index for March 2012 2012.)

Java

90-luvulla tuli tarve ohjelmointikielelle, jota voitaisiin käyttää kuluttajaelektroniikassa. Olemassa olevat ohjelmointikielet eivät olleet soveltuvia tähän tarkoitukseen. Kielen piti olla yksinkertainen ja luotettava ja sillä tehtyjen ohjelmien piti toimia uu-

sissa erilaisissa mikropiireissä ilman muutoksia. Tähän tarkoitukseen kehitettiin uusi ohjelmointikieli, Java. (The History of Java Technology 2012.)

Java ei kuitenkaan menestynyt kovin hyvin eletroniikkalaitteissa, ja kehitysryhmä suuntasi kieltä internetohjelmointiin. Java integroitiinkin Netscape Navigator internetselaimeen heti sen kehitysvaiheen alkupuolella. (The History of Java Technology 2012.) Tästä alkoi Javan suosio kohti suosituinta ohjelmointikieltä, jota se on tällä hetkellä (TIOBE Programming Community Index for March 2012 2012). Nykyisin Javan vahvuus on palvelinsovelluksissa. Java Enterprise Edition onkin tarkoitettu juuri tähän tarkoitukseen.

Java on täysin oliopohjainen ja laiteympäristöstä riippumaton. Sen uusia ominaisuuksia olio-ohjelmointikielissä olivat mm. roskienkeruu, joka vapauttaa muistia, kun sitä ei enää tarvita, sekä vahva tyypitys, eli jokaisella muuttujalla on tyyppi ja muuttuja voi saada ainoastaan tyypin mukaisia arvoja. Java on staattisesti tyypitetty, eli ohjelman tyypitykset tarkistetaan käännöksen aikana.

Javaan on suunnitteilla funktionaalisia ominaisuuksia tulevaisuudessa. Tällaisia ovat mm. closuret sekä nimettömät metodit (Java 8: New Feature 2012). Funktionaalisilla ominaisuuksilla pyritään helpottamaan rinnakkaisuutta eli ohjelman suoritusta usealla prosessorilla.

Poiketen tavanomaisista ohjelmointikielistä, Javan lähdekoodi käännetään tavukoodiksi, joka ajetaan JVM-virtuaalikoneessa eikä suoraan käyttöympäristössä.

Javaa käytettiin tässä opinnäytetyössä oliopohjaisena ja imperatiivisena vertailukielenä. Pääasiassa esimerkit ovat kuitenkin tehty Scala-ohjelmointikielellä. Jos esimerkin koodi on Javaa, se on kerrottu aina ennen esimerkkiä selvästi.

2.5 Funktionaalinen ohjelmointi

Funktionaalinen ohjelmointi on deklaritiivista, eli asiat esitetään niiden välisillä suhteilla, relaatioina. Funktionaalisessa ohjelmoinnissa ohjelma rakennetaan funktioista, jotka kutsuvat toisiaan. Funktionaalisen ohjelmoinnin funktio muistuttaa matematiikan funktiota. Funktionaalinen ohjelmointi onkin syntynyt ajatuksesta esittää ohjelmat matemaattisina määrittelyinä. Funktionaalisissa kielissä on paljon lambda-laskennan piirteitä, eli funktion arvo saadaan selville sieventämällä lauseketta säännöllisesti yksinkertaisemmaksi. Funktionaalisten kielten onkin sanottu olevan lambda-laskennan murteita. Funktionaaliset kielet sopivat siksi hyvin matemaattisiin ongelmiin.

Puhtaassa funktionaalisessa ohjelmassa kaikki sen toiminnot tehdään funktiokutsujen avulla. Funktiot muodostavat abstrakteja toimintoja. Funktioilla on argumentteja, joihin itse funktiota sovelletaan. Funktio ei pysty muuttamaan argumenttinsa arvoa. Funktio palauttaa tuloksen, joka on funktion arvo kyseisillä argumenteilla, tai uuden funktion. Puhtaassa funktionaalisessa ohjelmassa ei ole olemassa tilaa, eikä siten sijoituslauseita tai silmukoita, koska ei ole siis tilaa, jota muuttaa. Tästä johtuen funktiot ovat siis sivuvaikutuksettomia.

Puhtaan funktionaalisen ohjelman voidaan sanoa olevan laskutoimitusten sievennystä kuten lambda-laskennassakin. Useat funktionaaliset ohjelmointikielet eivät ole kuitenkaan puhtaasti funktionaalisia, ja ne voivat tukea muuttuvia muuttujia sekä sivuvaikutuksia. Funktion olisi kuitenkin hyvä palauttaa sama arvo tietyllä argumentilla tilasta riippumatta. Tätä kutsutaan läpinäkyvyydeksi. Ohjelmointikieltä sanotaan funktionaaliseksi, jos sillä ohjelmoidaan pääsääntöisesti kuten puhtaasti funktionaalisella kielellä. (Funktionaalinen ohjelmointi 2012.) Useissa funktionaalisissa kielissä on mukana myös proseduraalisia elementtejä.

Useat funktionaaliset ohjelmointikielet ovat laiskoja, eli funktion argumentit lasketaan, vasta kun niitä tarvitaan. Tämän avulla suorituskyky paranee, koska vältetään tarpeeton laskenta. Tämä mahdollistaa myös äärettömät tietorakenteet, joita voi-

daan rakentaa sitä mukaan, kun niitä tarvitaan. Tietorakenne on siis potentiaalisesti ääretön ohjelman saadessa käydä läpi vain äärellistä osaa. Tietorakenteet esitetään yleensä taulukoina tai listoina, joista voidaan viitata toisiin taulukoihin tai listoihin.

Käännetyn funktionaalisen kielen nopeus on nykyisin lähes sama kuin muiden ohjelmointiparadigmojen koodin. Binäärikoodit ovat myös lähes samankokoisia.(Funktionaalinen ohjelmointi 2012.) Eroa ei siis käännöksissä ja niiden ajamisessa niinkään tule.

Suurin ero puhtaalla funktionaalisella(deklaritiivinen) ja muilla (imperatiivinen) kielillä on tila, jota funktionaalisella ei ole siis olemassa.

Funktionaalisen ohjelmoinnin edut

- Korkea abstraktiotaso, joka tarkoittaa myös vähemmän koodia. Vähemmän koodia tarkoittaa vähemmän virheitä.

- Nopeus, koska vähemmän koodia tarkoittaa myös nopeampaa kehitystä.

- Uudelleenkäytettävyys, koska funktioilla ei ole sivuvaikutuksia ja funktioiden yhdistely on monipuolista. Ohjelma voidaan jakaa pienempiin ja yleiskäyttöisempiin osiin.

- Laajennettavuus, koska ohjelman osat eivät vaikuta toisiinsa.

- Selkeys, koska oikeanlaisella funktionaalisella ohjelmoinnilla saadaan selkeästi jäsenneltyä koodia.

- Induktio eli ohjelma voidaan kirjoittaa helposti vastaamaan määritystä ja todistaa oikeaksi induktiolla.

- Rinnakkaisuus, sillä funktiot ovat sivuvaikutuksettomia eli jaettavaa tilaa ei ole. Siksi erilliset funktiot voidaan laskea toisistaan riippumatta erillisissä prosessoreissa tai ytimissä, joten ohjelman kuorma voidaan jakaa tasapuolisesti niiden hoitettaviksi. Funktionaalista ohjelmointia sanotaan perinnöllisesti rinnakkaiseksi. Perinnöllinen siksi, että rinnakkaisuus johtuu paradigmasta itsestään. (Funktionaalinen ohjelmointi 1996.)

- Yksinkertainen syntaksi ja sen suuri ilmaisuvoima. Kirjoitustyyli on lähellä ongelman määrittelyä.

- Roskienkeruu, funktionaalisten kielten universaali ominaisuus on, että muistin varaus ja vapauttaminen hoidetaan automaattisesti.

- Funktiovalikoiman laajuus useissa kielissä.

- Helppo testattavuus, koska funktio palauttaa samalla argumentilla aina saman arvon.

- Ohjelmien täsmällinen määritys.

Funktionaalisen ohjelmoinnin haitat

- Ajatusmaailma, joka on hyvin erilainen kuin muissa ohjelmointiparadigmoissa. Sen oppiminen voi olla vaikeaa proseduraaliseen tai oliopohjaiseen ohjelmointiin tottuneelle. Puhtaassa funktionaalisuudessa ei ole esimerkiksi proseduraalisessa sekä oliopohjaisessa paljon käytettyjä toistorakenteita tai muuttuvia muuttujia.

- Monimutkainen arvojen syötön ja tulostuksen toteutus ohjelmassa. Koska sivuvaikutuksia ei sallita eikä muuttujia ole, on jouduttu keksimään toimintoja niiden toteutukseen. Esimerkiksi Haskell-ohjelmointikielessä on käsite mona-

di, jonka avulla syöttö ja tulostus hoidetaan.

- Reaalimaailman käsitteitä voi olla hankalaa ilmaista.

- Suorituskyky ei ole välttämättä yhtä tehokas jonkin ongelman ratkaisussa kuin se olisi muissa ohjelmointiparadigmoissa. Huonosti toteutettu funktionaalinen ohjelma voi olla hidas ja suuri kooltaan.

Funktionaalisen ohjelmoinnin suosio

Funktionaalisten ohjelmointikielten suosio on tällä hetkellä noin neljä prosenttia. Käytetyimpiä puhtaasti funktionaalisia ohjelmointikieliä ovat Lisp, Logo sekä Scheme. (TIOBE Programming Community Index for March 2012 2012.) Puhtaasti funktionaalisten kielten käyttö on tekoälytutkimuksen ulkopuolella harvinaista.

2.6 Scala

Scala on yleiskäyttöinen ohjelmointikieli, joka on suunniteltu ilmaisemaan yleisiä ohjelmointimalleja lyhyellä, elegantilla ja tyyppiturvallisella tavalla. Se yhdistää sujuvasti oliopohjaisten ja funktionaalisten kielten ominaisuuksia mahdollistaen paremman tuottavuuden. Koodin koko on tyypillisesti pienentynyt kahdes- kolmasosaan verrattuna vastaavaan Java-ohjelmaan. Monet yritykset, jotka ovat riippuvaisia liiketoimintakriittisistä ohjelmista, ovat kääntymässä Scalan puoleen edistääkseen ohjelmistokehityksensä tuottavuutta, ohjelmien skaalautuvuutta ja yleistä luotettavuutta. (Introducing Scala 2012.)

Scala on siis ohjelmointikieli, joka yhdistää edellä käydyistä ohjelmointiparadigmoista kaksi, funktionaalisuuden ja oliopohjaisuuden. Scalan kehittäminen alkoi vuonna 2001, ja se julkaistiin vuonna 2003. Nimi Scala tulee skaalautuvasta kielestä (Scalable language) tarkoittaen, että kieli on suunniteltu kasvamaan käyttäjiensä tarpeiden mukaan. (Odersky, Spoon & Venners 2010.)

Scala ei ole vielä kovin käytetty ohjelmointikieli, mutta monet uskovat Scalan ja funktionaalisen ohjelmoinnin lisääntyvän tulevaisuudessa. Scalan suurin referenssi on Twitter, joka on siirtynyt suurimmaksi osaksi Ruby-ohjelmointikielestä Scalan käyttöön (Twitter on Scala 2009). Opinnäytetyön toimeksiantaja otti Scala-kielen käyttöön osassa projektejaan vuonna 2011.

Scala on ympäristöriippumaton kuten Java, koska se on toteutettu myös JVM:n päälle. Scala tuottaa käännösvaiheessa samanlaista tavukoodia kuin Javakin ja ajetaan siis Java virtuaalikoneessa. Itse asiassa Scalassa voidaan käyttää kaikkia Javan kirjastoja hyväksi, vieläpä helpolla tavalla. Myös kaikkia muita Java-luokkia voidaan käyttää luonnollisesti. Scalaa voidaan käyttää samoihin tarkoituksiin kuin Javaakin, kuten kriittisiin ja suorituskykyä vaativiin järjestelmiin. (Ohjelmointikielen valinta 2012.)

Scala on vahvasti ja staattisesti tyypitetty kuten Javakin. Scala on myös toisaalta dynaaminen, sillä arvoille ei välttämättä tarvitse kertoa tyyppiä, vaan kääntäjä yrittää itse selvittää tyypin.

Scalassa funktiot ovat arvoja kuten numeeriset arvotkin, toisaalta kaikki arvot ovat myös olioita. Funktioita voidaan välittää parametreina, muuttujan arvona voi olla funktio, ja funktio voi palauttaa arvonaan funktion. Scalassa voidaan myös käyttää funktioliteraaleja, jotka ovat nimettömiä funktioita, yhtä lailla kuin numeerisia literaalejakin. Metodiksi Scalassa voidaan sanoa operaatiota, joka on luokan jäsen.

Scalaa voidaan kirjoittaa hyvin erilaisilla tavoilla. Java-ohjelmointikieltä osaavan on helppo siirtyä Scala-ohjelmointikieleen, koska sitä voidaan kirjoittaa hyvin samalla tavalla käyttäen Scalan oliopohjaisia ominaisuuksia. Scalan edistyksellisiä ja funktionaalisia ominaisuuksia ei tarvitse heti ottaa käyttöön, vaan niiden osuutta voi lisätä oman osaamisen kehityksen mukaan.

Scalan huonona puolena voidaan pitää sen huonoa taaksepäin yhteensopivuutta (Scala's version fragility make the Enterprise argument near impossible 2012).

3 FUNKTIONAALISET OMINAISUUDET

3.1 Sivuvaikutuksettomuus

Scalalla ensimmäinen asia on pyrkiä käyttämään ainoastaan val-tyyppisiä eli muuttumattomia muuttujia pyrittäessä funktionaaliseen sivuvaikutuksettomaan tyyliin. Muuttumaton muuttuja Scalassa luodaan käyttämällä sanaa val. Sitä voidaan verrata Javan final-määriteltyyn muuttujaan. Muuttuva muuttuja on puolestaan var, joka ei ole siis funktionaalinen.

Käsitellään var- ja val-tyyppiset muuttujat luokan luonnin yhteydessä yhden huomionarvoisen asian takia. Luodaan luokka Test:

```
class Test {
  var number = 0
}
```

Huomattakoon jo tässä vaiheessa että Scalassa ei tarvitse koodirivin loppuessa olla puolipistettä. Se voi olla siellä, mutta se ei ole pakollinen. Scala suosittelee jättämään ne pois. Luodaan luokasta nyt instanssi eli olio:

```
val test = new Test
```

Olio on val-määritelty, eli se on muuttumaton. Seuraava ei siis onnistuisi:

```
test = new Test  // ei käänny, koska olio on muuttumaton
```

Seuraava kuitenkin onnistuisi, koska luokan sisäinen muuttuja on määritelty muuttuvaksi muuttujaksi var! Olio voidaan siis määritellä muuttumattomaksi, mutta sen sisäinen muuttuja voi olla kuitenkin muuttuva.

```
test.number = 1  // kääntyy, koska muuttuja on muuttuva
```

Seuraavassa esimerkissä muutetaan Scalalla tehty imperatiivinen metodi funktionaalisemmaksi:

```
def printArgs(args: Array[String]): Unit = {
  var i = 0
  while (i < args.length) {
    println(args(i))
    i += 1
  }
}
```

Metodi luodaan käyttämällä sanaa ”def”. Metodin rakenne on siis ”def metodinNimi(parametrit) : palautusarvon tyyppi = {}”. Scalassa parametrit ovat muuttumattomia. Metodin palautusarvona esimerkissä on Unit, joka tarkoittaa, että metodi ei palauta mitään järkevää arvoa. Metodilla on kuitenkin aina jokin palautusarvo ja siten tyyppi. Tarkalleen ottaen esimerkki palauttaa Unit-tyyppisen ()-arvon, joka on tyhjä. Scalassa palautusarvon tyyppiä ei ole pakko määrittää vaan kääntäjän voi antaa selvittää se. Koodin selkeyden ja lukemisen kannalta voi olla kuitenkin hyvä määrittää se erikseen. Scalassa suositellaan jättämään return-sana pois, jolloin metodilla ei voi olla useata palautusarvoa. Tällä filosofialla rohkaistaan tekemään isoista metodeista pienempiä.

Muutetaan seuraavaksi koodia funktionaalisemmaksi poistamalla var-muuttuja:

```
def printArgs(args:Array[String]: Unit {
  for(arg <- args)
  println(arg)
}
```

Esimerkistä huomataan, että funktionaalisemmasta koodista on hyötyä. Koodi on selkeämpi, lyhyempi sekä vähemmän virheille herkkä. Metodi ei ole kuitenkaan puhtaasti funktionaalinen, koska sillä on sivuvaikutuksia. Metodin sivuvaikutus on tulos-

tus eli ”println”. Metodin palautusarvosta voidaan jo huomata sivuvaikutus, koska Unit tyyppihän ei palauta mitään järkevää arvoa. Koodin ainoa mahdollisuus vaikuttaa on siis jonkinlaisella sivuvaikutuksella. Funktionaalisempi lähestymistapa on määritellä metodi, joka muotoilee annetun taulukon, ja palauttaa sen tulostettavaksi:

```
def formatArgs(args: Array[String]) = {
  args.mkString(“\n”))
}
```

Nyt metodilla ei ole sivuvaikutuksia. Metodia ”mkString” voidaan kutsua kaikille iteroitaville kokoelmille kuten taulukoille, listoille, seteille ja mapeille. Kyseinen metodi palauttaa merkkijonon, joka sisältää kokoelman eroteltuna annetulla merkkijonolla. Esimerkin “\n” erottelisi siis kokoelman “nolla”, “yksi”, “kaksi” seuraavanlaiseksi: “nolla\nyksi\nkaksi”. Esimerkkihän ei nyt tulosta mitään kuten aiempi esimerkki, mutta nyt puolestaan sen palautusarvo voidaan tulostaa helposti:

```
println(formatArgs(args))
```

Funktionaalisessa tyylissä pyritään suunnittelemaan mahdollisimman sivuvaikutuksetonta koodia. Sivuvaikutuksilta ei voida täysin välttyä, koska silloin ohjelma ei voisi esimerkiksi ottaa arvoja vastaan tai tulostaa niitä.

Scala-ohjelmointikielen kehittäjät suosittelevat sivuvaikutuksetonta ohjelmointityyliä, mutta myös sivuvaikutukset sallitaan. “Tavoittele mieluummin val-muuttujia, muuttumattomia olioita ja sivuvaikutuksettomia metodeja. Käytä var-muuttujia, muuttuvia olioita ja sivuvaikutuksellisia metodeja, kun on erityinen tarve ja peruste niiden käytölle.” (Odersky, Spoon & Venners 2010, 54)

Seuraavassa esimerkissä lasketaan listan lukujen keskiarvo imperatiivisesti:

```
def average(n: Int): List[Int] = {
  var sum = 0
```

```
  val listSize = n.length
  for(i <- 0 to listSize - 1) {
    sum += n(i)
  }
  sum / listSize
}

val result = average(List[Int](1,2,3))
```

Koodin rakenne on imperatiiviselle ohjelmoinnille yleinen, eli se sisältää toistorakenteen sekä apumuuttujan. Jokaisella for-silmukan kierroksella apumuuttuja “sum” muuttuu. Huomattakoon, että funktiot palauttavat viimeisen rivin arvon ilman return-sanaa. Scalassahan se on mahdollista jättää kirjoittamatta, jolloin palautetaan automaattisesti viimeisen rivin arvo. Muutetaan koodi funktionaaliseksi:

```
def average(n: List[Int]): Int = {
  val sum = n.reduceLeft((a:Int, b:Int) => a + b)
  sum / n.size
}

val result = average(List[Int](1,2,9,4,4,6,6))
```

Nyt ei ole enää muuttuvaa apumuuttujaa eikä toistorakennetta. Funktio ”reduceLeft” käy listan läpi vasemmalta oikealle. Funktiolle määritellään kaksi arvoa: ”a” ja ”b”, joita käytetään laskennassa. Ensin lasketaan 1 + 2 = 3 ja sitten 3 + 9 = 12 ja sitten 12 + 4 = 16 jne. Lopuksi jaetaan summa listan koolla, eli numeroiden määrällä, jolloin saadaan keskiarvo.

Esimerkin muuttuja ”sum” olisi myös mahdollista määritellä vielä lyhemmin:

```
val sum = n.reduceLeft(_ + _)
```

Myöhemmin tässä opinnäytetyössä selviää, miten tämä on mahdollista.

3.2 Rekursio

Funktionaalisen ohjelmoinnin perusmekanismi on funktiokutsu, joka palauttaa aina jonkin arvon. Silmukoita ei voida täten käyttää puhtaassa funktionaalisessa ohjelmoinnissa, koska ne eivät palauta mitään arvoa. Toisto suoritetaan täten rekursion avulla.

For-toistorakenne

Funktionaalisessa ohjelmoinnissa ei ole olemassa ollenkaan siis for-toistorakennetta. Sen sijaan käytetään rekursiota, joka perustuu funktion uudelleen kutsumiseen funktion sisältä.

Luodaan Javalla for-toistorakenne eli silmukka, joka toistuu viisi kertaa:

```
for(int i = 0; i < 5; i++) {
  // Do something...
}
```

Scalalla sama tehtäisiin funktionaalisesti foreach-metodilla, joka on rekursiivinen:

```
1 to 5 foreach( x => // Do something... )
```

Scala-versiossa ei siis luoda kokonaislukumuuttujaa i ollenkaan.

Otetaan toinen esimerkki for-silmukan käytöstä, jossa käydään taulukko läpi ja tulostetaan sen sisältö Javalla:

```
for(String arg : args) {
  System.out.println(arg);
}
```

Scalassa taulukko voidaan käydä läpi samalla foreach-metodilla, kuten edellisessä

Scalalla luodussa esimerkissä:

```
args.foreach(arg => println(arg))
```

Taulukon läpikäyminen tehdään metodin ”foreach” avulla, joka saa parametrina literaalifunktion, jota se soveltaa jokaiselle taulukon alkiolle. Literaalifunktio saa siis argumentin ”arg”, jota käytetään sen metodissa ”println()”. Literaalifunktio käydään läpi tarkemmin luvussa 3.3.1, Literaalifunktio ja sen asettaminen muuttujaan.

Scalassa edellinen esimerkki voidaan kirjoittaa itse asiassa vielä lyhyemmällä tavalla:

```
args.foreach(println)
```

Jos funktion literaali koostuu yhdestä lauseesta, joka ottaa vain yhden argumentin, niin silloin ei tarvitse erikseen nimetä ja määritellä argumenttia. Tätä kutsutaan osittain sovelletuksi funktioksi. Myös seuraava on mahdollista:

```
args foreach(println)
```

Scalassa ei ole pakollista käyttää pistettä kutsuttaessa metodia, jos se ottaa ainoastaan yhden parametrin. Jatkossa tullaan käyttämään kuitenkin pistettä metodien kutsuissa, sillä se helpottaa koodin lukemista.

Scalassa myös operaattorit ovat metodeja eli yhteenlasku 1 + 2 voidaan myös esittää ”1.+(2)”. Oliolla 1 on metodi +, joka ottaa parametrina arvon 2 ja palauttaa näiden yhteenlasketun summan. Numerot ovat siis Scalassa olioita, joilla on normaaliin tapaan metodeja, joita ne voivat kutsua. Muistettakoon, että Scalassa kaikki arvot ovat olioita! Teknisesti Scalassa ei ole olemassa myöskään operaattoreita, vaan ne kaikki ovat metodeja.

while-silmukka

Myöskään while-silmukkaa ei ole mahdollista käyttää funktionaalisuudessa, sillä sekään ei palauta mitään arvoa. Seuraavassa esimerkissä on Javalla luotu metodi, joka käyttää while-silmukkaa suurimman yhteisen jakajan laskemisessa:

```
private long gcd(long x, long y) {
 long a = x;
 long b = y;
 while(a != 0) {
   long temp = a;
   a = b % a;
   b = temp;
 }
 return b;
}
```

Rekursiivisesti sama voitaisiin kirjoittaa Scalalla seuraavanlaisesti:

```
def gcd(x: Long, y: Long): Long {
 if(y == 0) x else gcd(y, x % y)
}
```

Nyt if-lauseella tarkistetaan aina ensin onko muuttuja ”y” nolla, jolloin palautetaan muuttuja ”x”. Muuten kutsutaan funktiota uudelleen. Funktiota kutsutaan siis niin pitkään, kunnes muuttujan ”x” arvoksi tulee nolla.

Scalassa on kuitenkin olemassa sekä for- että while-silmukka, mutta niiden käyttöä suositellaan välttämään, jotta saadaan funktionaalisempaa koodia.

3.2 Muuttumattomat tietorakenteet

3.2.1 Lista

Lista on funktionaalisen ohjelmoinnin yleisin tietorakenne, jonka alkioita käsitellään funktioilla. Lista sisältää elementtejä, joiden tyyppi täytyy kuitenkin olla sama. Eli jos lista sisältää numeron, jonka tyyppi on ”Integer”, myös muiden listan elementtien täytyy olla samantyyppisiä. Sekä Javasta että Scalasta löytyy lista, joita kuitenkin käytetään hiukan eri tavalla. Listat ovat tärkeä osa funktionaalista ohjelmointia, sillä usein funktiolle viedään lista, jota käydään läpi, ja palautusarvona on alkuperäisen listan osalista.

Elementin lisäys listaan Javalla tapahtuisi seuraavasti:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(2);
list.add(3);
list.add(0,1);
```

Ensiksi luodaan lista, johon lisätään kolme elementtiä. Viimeinen elementti halutaan asettaa listan ensimmäiseksi eli listan paikkaan 0.

Scalalla listaan lisätään elementti listan alkuun seuraavasti:

```
val list1 = List(2,3)
val list2 = 1 :: list1
```

Ensimmäiseksi luodaan lista ”list1”, joka sisältää kaksi elementtiä. Scalan kääntäjä ymmärtää automaattisesti listan tyypiksi ”Integerin”, koska se sisältää kokonaislukuja. Seuraavaksi luodaan uusi lista, jolle annetaan yksi elementti sekä ensiksi luotu lista ”list1” käyttämällä ”::” operaattoria, joka on ehkä yleisin operaattori listojen käytössä (”::” on siis oikeasti uuden listan palauttava metodi, koska Scalassa ei ole operaattoreita). Huomattakoon, että vanhaan listaan ei siis lisätä uutta elementtiä,

vaan luodaan uusi lista, joka sisältää vanhan listan. Funktionaalisuudessahan ei ole muuttuvia muuttujia eikä siten muuttuvia tietorakenteita! Toinen huomio ”::”- metodiin liittyen on, että se suoritetaan sen oikealla puolella olevaan muuttujaan. Normaalisti se suoritettaisiin sen vasemmalla puolella olevaan muuttujaan, mutta Scalassa kaksoispisteeseen päättyvä metodi suoritetaan oikeanpuoleiseen muuttujaan. Siksi se voitaisiin esittää myös:

```
val list2 = list1.::(1)
```

Lista voitaisiin myös tehdä muuttuvaksi:

```
var list = List(2,3)
list = 1::list
```

Listaa ei kuitenkaan muuteta, vaan lista ”list” on uusi lista! ”1::list” palauttaa uuden listan, joka on siis ”List(1,2,3)”.

Suurin ero Javan ja Scalan listoilla on, että Scalan lista on muuttumaton, joka mahdollistaa funktionaalisuuden. Scalalla kylläkin voidaan tehdä oikeasti muuttuva lista, mutta silloin se pitää erikseen tuoda ”import”-lauseella, jonka jälkeen sen käyttö on kuitenkin samanlaista. Muuttuva lista ei ole kuitenkaan enää funktionaalista, joten sitä ei suositella käytettävän.

3.2.2 Tuple

Tuplet ovat kuten listoja, mutta voivat sisältää erityyppisiä elementtejä. Tuplet ovat yleisiä funktionaalisissa kielissä, mutta oliopohjaisessa Javassa niitä ei ole. Tuplet ovat käteviä, jos esimerkiksi funktion pitää palauttaa useita erilaisia arvoja. Tupleja on hyvä käyttää myös tietorakenteena, jos tietää etukäteen kuinka monta elementtiä tietorakenteen tulee sisältää.

Scalassa tuplen luonti käy seuraavasti:

```
val pair = (1, “testi”)
```

Ensimmäisen elementin tyyppi on ”Integer”, jonka arvoksi on asetettu ”1” ja toisen elementin tyyppi on ”String”, jonka arvoksi on asetettu “testi”. Tuplen osat voi tulostaa seuraavalla tavalla:

```
println(pair._1, pair._2)
```

Esimerkin tuple “pair” on Tuple2. Scala määrittää sen sellaiseksi, koska Tuple2 on ”[Int,String]” eli sillä on kaksi elementtiä. Tuple4 olisi esimerkiksi ”[Int,String,Boolean,Int]”, koska sillä on neljä elementtiä. Scalassa on määritelty tupleja 22 kappaletta, vaikka sinällänsä niitä voisi olla ääretön määrä. Niitä kaikkia vain ei ole määritelty.

3.2.3 Map

Myös map-tietorakenne on tärkeä funktionaalisessa ohjelmoinnissa listojen lisäksi. Map on tietorakenne, joka sisältää avain-arvo pareja. Jos avaimen määritellään olevan ”Integer”-tyyppinen, kaikkien avaimien pitää olla sellaisia. Sama pätee arvoille.

Luodaan Javalla map, johon lisätään kaksi avain-arvo paria:

```
Map<String,Integer> map = new HashMap<String,Integer>();
map.put(“yksi”,1);
map.put(“kaksi”,2);
```

Scalalla map luodaan seuraavanlaisesti:

```
val map = Map(“yksi” -> 1, “kaksi” -> 2)
```

Scalan kääntäjä osaa automaattisesti määritellä mapin avaimen ”String”-tyyppiseksi sekä arvon ”Integer”-tyyppiseksi. Mapin muuttumissäännöt ovat samanlaiset kuin listassakin. Mappiin voidaan lisätä avain-arvo pareja ”+=”-metodilla, joka palauttaa

aina uuden mapin:

```
map += (“kolme” -> 3)  // ei toimi koska map on val
```

Vaikka edellä oleva rivi ei käännykään, siinä on mielenkiintoinen metodi ”->”. Sama rivi voitaisiin kirjoittaa myös:

```
map += (“kolme”).->(3)
```

String kutsuu siis metodia ”->”, jolle se antaa parametrina kokonaisluvun ”3”. Metodi ”->” itse asiassa palauttaa tuplen, tässä tapauksessa tuplen ”Tuple2[String,Integer]”, joka siis sisältää avaimen ja arvon. Lopulta kutsutaan metodia ”+=”, jonka parametrina on kyseinen tuple. Metodia ”->” voidaan kutsua kaikille Scalan olioille. Tällaista metodia kutsutaan implisiittiseksi muuntamiseksi.

3.3 Ensimmäisen luokan funktiot

Funktionaalisessa ohjelmoinnissa on olemassa niin sanottuja ensimmäisen luokan funktioita (first-class function). Näillä tarkoitetaan funktion käyttämistä argumenttina, funktion palauttamista funktion arvona tai funktion asettamista muuttujaan. Myös literaalifunktiot eli nimettömät funktiot ovat tällaisia.

3.3.1 Literaalifunktio ja sen asettaminen muuttujaan

Scalassa funktioliteraali käännetään luokaksi, joka on ajonaikana kuitenkin funktion arvo. Funktioliteraalin ja arvon ero on, että funktioliteraali on olemassa koodissa, kun funktion arvo on olemassa ajonaikana oliona muistissa. Eron voi sanoa olevan hiukan samanlainen kuten luokalla, joka on koodissa ja luokan oliolla, joka on olemassa ajonaikana muistissa. Scalassa jokainen funktio kuuluu johonkin luokkaan. Funktioliteraali ilman parametreja kuuluu luokkaan ”Function0”, funktioliteraali yhdellä parametrilla kuuluu luokkaan ”Function1” ja niin edelleen.

Funktion arvot ovat siis olioita, joten niitä voidaan asettaa muuttujaan:

```
val plusOne = (x: Int) => x + 1
```

Tämän funktion arvo kuuluu luokkaan ”Function1”, koska sillä on yksi parametri. Funktion arvot ovat myös funktioita, joten niitä voidaan kutsua normaalisti:

```
plusOne(2)
```

Palautusarvona olisi siis 2 + 1 eli 3.

Funktioliteraali toimii kuten normaali funktio eli se palauttaa viimeisen rivin arvon. Siksi funktiossa voidaan tehdä myös muutakin:

```
val plusOne = (x: Int) => {
 println(“Literaalifunktion sisällä”)
 x + 1
}
```

Luvussa 3.1 käydyssä esimerkissä taulukon ”foreach”-metodi otti literaalifunktion parametrinaan:

```
args.foreach(arg => println(arg))
```

Tätä ”foreach”-metodia voidaan käyttää kaikille Collection-luokan olioille, jolla voidaan käydä helposti olion sisältämät elementit läpi. Collection-luokalla on myös käytännöllinen metodi ”filter”:

```
args.filter((x: String) => x == “test”)
```

Tämä funktioliteraali palauttaa taulukosta ainoastaan elementit, joiden sisältö on “test”. Tyypin määritys arvolle ”x” ei ole pakollista, sillä kääntäjä ymmärtää tyypin, koska args-taulukko sisältää ”String”-tyyppisiä elementtejä. Sama voidaan siis kirjoittaa myös seuraavanlaisesti:

```
args.filter(x => x == “test”)
```

Paikanpitäjä

Esimerkin koodi on mahdollista muuttaa vieläkin lyhyemmäksi Scalan paikanpitäjä (placeholder) syntaksilla. Paikanpitäjän merkintä on alaviiva. Sitä voidaan käyttää, jos parametria käytetään vain kerran literaalifunktiossa.

```
args.filter(_ == “test”)
```

”Filter”-metodi korvaa nyt alaviivan taulukon sen hetkisellä elementillä.

Literaalifuntioiden yhdistäminen

Literaalifunktioita voidaan myös yhdistellä:

```
def main(args: Array[String]): Unit = {
 args.filter(arg => arg.startsWith("G")
  .foreach(arg => Console.println("Löydettiin: " + arg)
}
```

Esimerkissä suodatetaan ”filter”-metodilla ja sen literaalifunktiolla taulukosta ”G”-kirjaimella alkavat merkkijonot, jotka tallentuvat uuteen taulukkoon. Taulukko käydään läpi ”foreach”-metodilla ja sen literaalifunktiolla tulostetaan löydetyt merkkijonot. ”Filter”-metodin määrittelystä voidaan tarkastaa, että se palauttaa uuden taulukon, jota voidaan siten käyttää toisessa ”foreach”-metodissa.

```
def filter (p : (A) => Boolean) : Array[A]
```

”Filter”-metodi määrittelee, että ”se palauttaa taulukon, joka sisältää kaikki elementit tyyppiä A, jotka toteuttavat predikaatin p tyyppiä A”. Esimerkissä ei ole erikseen määritelty taulukon sisältämiä elementtejä merkkijonoiksi ”filter”-metodissa, sillä

kääntäjä ymmärtää ne jo sellaisiksi, koska ”args”-taulukko on määritelty sisältävän merkkijonoja.

3.3.2 Funktio funktion argumenttina

Funktiolle voidaan antaa funktio argumenttina:

```
def functionExample(functionArgument: () => Unit): Unit =
{
 println(“Print from method”)
 functionArgument(“Print this too.”)
}

def printLine(a: String): Unit = {
 println(a)
}

def main(args: Array[String]): Unit = {
 functionExample(printLine)
}
```

Esimerkin “Main”-metodissa kutsutaan funktiota “functionExample”, joka ottaa argumenttinaan toisen funktion, tässä tapauksessa funktion “printLine”. Jos funktion argumenttina on funktio, määritellään sille normaalisti nimi ja merkitään sen tyypiksi ”()”. Tällöin funktioargumenttia voidaan kutsua normaalisti funktiossa tällä funktion nimellä.

3.3.3 Funktion palautusarvona funktio

Funktio voi myös palauttaa funktion, kun sen palautustyypiksi asetetaan ”()”:

```
def function(): () => String {
 (() => {
   "Something"
 })
}
```

Funktio siis palauttaa nyt funktion ”()”, joka palauttaa merkkijonon “Something”. Funktio voitaisiin asettaa muuttujaan ja tulostaa se.

```
val a = function()
println(a())
```

3.3.4 Sisäkkäiset funktiot

Luokkien ja objektien apufunktioita usein halutaan piilottaa, jotta niihin ei päästäisi käsiksi luokan ulkopuolelta, sillä niitä on harvoin järkevää käyttää sellaisenaan.

Luodaan objekti ”LongLines”, jolla on kaksi funktiota, joista toinen on ”private”-määritelty eli siihen ei päästä käsiksi objektin ulkopuolelta. ”Private”-määrittely toimii samalla tavalla Scalassa kuin Javassakin.

```
object LongLines {
 def processFile(filename: String, width: Int) {
   val source = Source.fromFile(filename)
   for (line <- source.getLines())
     processLine(filename, width, line)
 }
 private def processLine(filename: String, width: Int,
     line:String) {
 if (line.length > width)
   // Do something
 }
}
```

”Private”-määritelty funktio voidaan myös siirtää ”processFile”-funktion sisäkkäiseksi funktioksi eli lokaaliksi funktioksi:

```
def processFile(filename: String, width: Int) {
 def processLine(line: String) {
   if (line.length > width)
     // Do something
 }
 val source = Source.fromFile(filename)
 for (line <- source.getLines())
   processLine(line)
}
```

Nyt tiedoston nimeä eli ”filename”-parametria käytetään myös sisäkkäisessä funktiossa sellaisenaan, eikä sitä tarvitse sille erikseen viedä parametrina.

3.4 Osittain sovellettu funktio

Osittain sovellettu funktio on lauseke, jossa funktiolle ei anneta kaikkia sen tarvitsemia argumentteja. Funktiolle voidaan olla antamatta yhtään argumenttia tai vain osa niistä. Luodaan funktio ”sum” ja asetetaan se muuttujaan ”a”:

```
def sum(a: Int, b: Int, c:Int) = {
 a + b + c
}

val a = sum _
```

Scala kääntäjä nyt määrittää muuttujan ”a” seuraavanlaiseksi:

```
a: (Int, Int, Int) => Int = <function3>
```

Muuttujan arvona on siis funktio, joka ottaa kolme kokonaislukua argumentteina,

jotka puuttuivat osittain sovelletusta funktiosta, joka siis määriteltiin ”sum _”. Se myös asettaa viittauksen uuteen funktion arvoon ”<function3>”, koska ”<function3>” määrittelee kolmiargumenttisen ”apply”-metodin. Muuttujalle voidaan antaa nyt kolme argumenttia ja se palauttaa arvonaan 1 + 2 + 3 eli 6:

```
a(1, 2, 3)
```

Muuttuja siis viittaa johonkin funktion arvoon. Scala kääntäjä luo automaattisesti funktion arvon, joka kuuluu sellaiseen funktioluokkaan, joka perii luokan tapaisen traitin eli piirreluokan ”Function3”. Piirreluokkia voi hiukan verrata Javan rajapintoihin. Piirreluokilla voi kuitenkin olla toteutettuja metodeita. ”Function3”-piirreluokalla on ”apply”-metodi, joka ottaa kolme argumenttia. Jos funktiosta puuttuisi kaksi argumenttia, perittäisiin luokka ”Function2”, jolla on ”apply”-metodi, joka ottaa kaksi argumenttia jne. Sama koodi voitaisiin siis kirjoittaa myös:

```
a.apply(1, 2, 3)
```

”Apply”-metodi tässä tapauksessa kutsuu ”sum(1, 2, 3)” ja palauttaa ”sum”-funktion arvon. Osittain sovellettu funktio voi myös ottaa osan tarvittavista argumenteista:

```
val b = sum(1, _: Int, 3)
```

Kääntäjä määrittäisi nyt muuttujan ”b” seuraavanlaisesti:

```
b: (Int) => Int = <function1>
```

Funktiosta puuttui siis nyt yksi argumentti ja siksi viitataan funktion arvoon ”<function1>”, joka kuuluu funktioluokkaan, jolla on ”apply”-metodi, joka ottaa yhden argumentin. Asetetaan muuttujalle ”b” argumentti ”5”:

```
b(5)
```

Tässä tapauksessa kutsutaan muuttujan ”apply”-metodia, joka kutsuu ”sum(1, 5, 3)”. Palautusarvona olisi nyt 1 + 5 + 3 eli 9.

3.5 Closure

Closure on funktion arvo, joka käyttää funktion ulkopuolista muuttujaa:

```
val number = 2
val addNumber = (x: Int) => x + number
```

Funktion arvo ”addNumber” on siis closure, koska se käyttää funktion ulkopuolista muuttujaa ”number”.

Closure pystyy myös muuttamaan ulkopuolista muuttujaa:

```
val numbers = List(1,2,3)
var sum = 0
numbers.foreach(sum += _)
```

Nyt muuttujan ”sum” arvo on 6.

3.6 Currying

Curried-funktiossa funktiota sovelletaan jokaiselle argumentille erikseen. Kun funktiolle viedään useita argumentteja, niin funktiota sovelletaan ensimmäiseen argumenttiin. Arvona saadaan uusi funktio, jota sovelletaan seuraavaan argumenttiin. Tätä jatketaan niin kauan, kunnes kaikki argumentit on käyty läpi.

Luodaan normaalisti kirjoitettu funktio, joka laskee kaksi lukua yhteen:

```
def sum(x: Int, y: Int): Int = {
  x + y
}
```

Muutetaan funktio curried-funktioksi:

```
def sum(x: Int)(y: Int): Int {
  x + y
}
```

Nyt funktiolla on kaksi erillistä argumenttilistaa, johon funktiokutsua sovelletaan. Funktiota voidaan käyttää nyt antamalla funktiolle kaksi erillistä argumenttia:

```
sum(1)(2)
```

”Sum”-funktiota kutsuttaessa kutsutaan siis samaa funktiota kaksi kertaa. Ensin kutsutaan ”sum”-funktiota ”x”-argumentilla, joka palauttaa funktioarvon seuraavalle funktiolle. Seuraava funktio kutsuu ”sum”-funktiota ”y”-argumentilla.

Curried-funktiota voidaan käyttää osittain sovelletussa funktiossa:

```
val onePlus = sum(1)_
onePlus(2)
```

Tässä tapauksessa palautetaan kokonaisluku kolme. Osittain sovellettua funktiota on helppo muuttaa, jotta voidaan lisätä esimerkiksi luku viisi:

```
val fivePlus = sum(5)_
fivePlus(2)
```

Nyt ”fivePlus”-funktio palauttaa kokonaisluvun seitsemän.

3.7 Pattern matching

Scalan pattern matchingia voidaan verrata Javan switch caseen. Scalan match-vertailu kuitenkin palauttaa aina arvon. Scalassa ei myöskään mennä enää eteenpäin seuraavaan vertailuun, jos ehto toteutuu toisin kuin Javassa. Scalassa jonkin ehdon on myös pakko toteutua tai aiheutuu ”MatchError”-poikkeus. Siksi pitää aina muistaa käydä läpi kaikki mahdolliset tapaukset. Tapaukset, joissa erikseen määritellyt tapaukset eivät toteudu, voidaan ottaa kiinni ”_” -merkinnällä. Merkintää sanotaan tässä tapauksessa niin sanotuksi villiksi kortiksi. Pattern match tehdään Scalalla seuraavanlaisesti:

```
def numberTest(x: Int): String = x match {
  case 1 => "yksi"
  case 2 => "kaksi"
  case _ => "muu numero"
}

println(numberTest(2))
```

Esimerkki ottaa argumenttinaan kokonaisluvun ja palauttaa numeroa vastaavan merkkijonon. Jos kumpikaan tapaus ”case 1” tai ”case 2” ei toteudu, palautetaan “muu numero”. Esimerkki tulostaa “kaksi”, koska numero kaksi toteutuu tapauksessa case 2.

Pattern match -funktion argumenttien ei ole pakko olla vain tietyn tyyppisiä, vaan ne voidaan määritellä ”Any”-tyyppisiksi. Tällöin voidaan vertailla kaikentyyppisiä muuttujia. Myös palautusarvoksi voidaan määritellä ”Any”, jolloin voidaan myös palauttaa kaikentyyppisiä arvoja. ”Any” on Scalan luokkahierarkkian juuriluokka. Kaikki luokat siis periytyvät ”Any”-luokasta. Luodaan vertailu, jonka argumentti sekä palautusarvo voi olla minkätyyppinen tahansa:

```
def numberTest(x: Any): Any = x match {
  case 1 => "yksi"
```

```
  case "kaksi" => 2
  case n: Boolean => "Tyyppina oli Boolean”
  _ => “muu”
}

println(numberTest(false))
println(numberTest(“kaksi”))
```

Esimerkki tulostaa ensimmäiselle riville “Tyyppinä oli Boolean”, koska tapaus ”n: Boolean” toteutuu ja toiselle riville 2, koska tapaus “kaksi” toteutuu ja funktio palauttaa tällöin kokonaisluvun kaksi.

3.8 Laiskat muuttujat

Laiskoilla muuttujilla tarkoitetaan muuttujia, jotka alustetaan vasta niitä tarvittaessa. Joissakin ohjelmointikielissä, kuten Haskell, kaikki arvot ja parametrit alustetaan laiskasti. Scalassa muuttuja voidaan osoittaa laiskaksi ”lazy”-määrityksellä. Laiskoilla muuttujilla voidaan ohjelman suoritusta tehostaa, sillä raskaita ja turhia operaatioita, joita ei välttämättä ohjelman suorituksen aikana tulla tarvitsemaan, ei tarvitse alustaa.

Seuraavassa luodaan kaksi objektia, joista toinen sisältää normaalin muuttujan ja toinen laiskan muuttujan:

```
object Test {
 val x = 2
}

object LazyTest {
 lazy val x = 2
}
```

Kun viitataan objektiin ”Test”, kääntäjä ilmoittaa “initializing x” eli muuttuja ”x” alus-

tetaan. Jos viitataan objektiin ”LazyTest”, tälläista kääntäjän ilmoitusta ei tule, eli muuttujaa ei alusteta vielä tässä vaiheessa. Kääntäjä kuitenkin ilmoittaa “initializing x” viitattaessa objektin muuttujaan ”LazyTest.x”, jolloin tapahtuu ”x”-muuttujan alustus.

Laiskoilla muuttujilla mahdollistetaan myös niin sanottu memoisaatio sekä äärettömät tietorakenteet. Memoisaatiossa funktion arvo tietyillä parametreillä tallennetaan muistiin, josta se voidaan hakea nopeasti uudelleen käyttöä varten.

4 FUNKTIONAALISTEN OMINAISUUKSIEN HYÖDYT

4.1 Lyhyempää ja tuottavampaa koodia

Tämän opinnäytetyön esimerkeistä voidaan huomata, että funktionaalisesti kirjoitetut metodit ovat lyhyempiä kuin imperatiivisesti kirjoitetut. Useissa tutkimuksissa on osoitettu, että keskimääräisesti ohjelmoija tuottaa koodirivejä saman verran kaikissa ohjelmointikielissä. (Functional programming 2012.) Tuottavuus siis paranee, mitä vähemmän koodia on tarvetta tehdä.

Seuraava esimerkki osoittaa kuinka funktionaalisesti kirjoitettu Scalalla tehty versio on Javalla kirjoitettua imperatiivista versiota paljon lyhyempi. Esimerkissä etsitään merkkijonosta isoja kirjaimia. Jos iso kirjain löytyy, muutetaan ”boolean”-arvo todeksi. Javalla tehty imperatiivinen versio:

```
boolean nameHasUpperCase = false;
for (int i = 0; i < name.length(); i++) {
  if(Character.isUpperCase(name.charAt(i))) {
    nameHasUpperCase = true;
    break;
  }
}
```

Sama voidaan kirjoittaa Scalalla funktionaalisesti yhdelle riville:

```
val nameHasUppercase = name.exists(_.isUpper)
```

Scalassa on laaja valikoima käteviä funktioita, joita voidaan soveltaa olioille. Varsinkin listoille, seteille ja mapeille löytyy hyvin kattava valikoima funktioita, joilla voidaan käydä tietorakennetta läpi. Näiden funktioiden käyttö yleensä vähentää koodin määrää huomattavasti, kuten edellisestäkin esimerkistä huomataan.

4.2 Turvallisuus

Sivuvaikutuksetonta koodia on helpompi ymmärtää. Jos luokka on muuttumaton, sillä ei ole myöskään olemassa tilaa. Näinollen voidaan olla varmoja esimerkiksi muuttujan arvosta eikä koodia tarvitse käydä läpi varmistuakseen siitä.

Muuttumattoman luokan oliota on helppoa ja turvallista käyttää koodissa eikä siitä tarvitse tehdä mahdollista kopiota, jossa voitaisiin varmistua sen oikeasta tilasta. Samanaikaiseet säikeet eivät voi myöskään rikkoa olion tilaa eikä säikeitä tarvitse lukita. Muuttumattomasta oliosta on myös turvallista muodostaa hajautustaulun avain, sillä sen arvo ei voi muuttua. Jos muuttuvan olion tilaa muutetaan, sitä ei esimerkiksi välttämättä enää löydetä silppujoukosta.

Toisaalta muuttumattomien luokkien olioista voidaan joutua joskus tekemään paljon kopioita, jolloin suorituskyky kärsii. Scalassa onkin olemassa monista muuttumattomista luokista muuttuvia versioita kuten String-luokan muuttuva versio StringBuilder-luokka.

Sivuvaikutuksettomat funktiot vähentävät virheiden määrää. Funktio palauttaa aina saman arvon ohjelman tilasta riippumatta. Funktiota voidaan kutsua monta kertaa peräkkäin vaikka eri säikeistä ja silti funktio palauttaa aina saman arvon. Funktiosta nähdään myös helposti mitä se ottaa vastaan argumentteinaan ja mitä se palauttaa arvonaan.

Seuraava esimerkki on osoitus siitä, kuinka Scalalla voidaan tehdä konkreettisesti turvallisia muuttujia funktionaalisesti. Esimerkin muuttuja ”half” on muuttujan ”n” puolet luvusta, jos luku on parillinen. Muuten aiheutuu poikkeus.

```
val half =
 if (n % 2 == 0)
   n / 2
 else
```

```
throw new RuntimeException(“n täytyy olla parillinen”)
```

”Throw”-metodi palauttaa aina tyypin ”Nothing” eli esimerkissä muuttuja ”half” on ”Nothing”, jos luku ei ole parillinen. Tästä hyödytään, jos yritetään käyttää tätä muuttujaa muualla. Tällöin ei voida vahingoittaa muuta koodia, koska arvoa ei voida käyttää sen ollessa ”Nothing”.

4.3 Helppo testattavuus

Puhtaan funktionaalisen funktion etuna on helppo testattavuus. Funktiot ovat kuin matemaattisia funktioita, jotka palauttavat aina saman arvon tilasta riippumatta.

Aikaisemmin opinnäytetyössä käytiin esimerkki läpi, jossa muutettiin metodi funktionaalisemmaksi, ja lopulta päädyttiin funktioon:

```
def formatArgs(args: Array[String]) =
args.mkString(“\n”))
```

Tällainen funktio on nyt helppo testata:

```
val result = formatArgs(Array(“yksi”,”kaksi”, “kolme”))
assert(result == “yksi\nkaksi\nkolme”)
```

”Assert”-metodi tarkistaa onko muuttuja “result” sama kuin “yksi\nkaksi\nkolme”, ja palauttaa ”boolean”-tyyppisen arvon. Jos tulos oli sama, palautetaan ”true”. Jos tulos ei ollut sama, palautetaan ”false”, ja aiheutuu ”AssertionError”-poikkeus.

4.4 Uudelleenkäytettävyys ja laajennettavuus

Korkeamman tason funktiot mahdollistavat sivuvaikutuksettomien funktioiden yhdistelemisen monelle tavalla. Funktioita voidaan myös asettaa muuttujiin. Tällaisilla funktioilla voidaan ohjelma jakaa pienempiin ja yleiskäyttöisempiin osiin, jotka kut-

suvat toisiaan. Currying on hyvä esimerkki siitä, kuinka funktio voidaan tehdä yleiskäyttöisemmäksi.

Luokat ja objektit siis sisältävät useita pieniä funktioita, jotka tekevät yhden hyvin määritellyn tehtävän. Näitä funktioita voidaan siten uudelleenkäyttää helposti. Ohjelman osien voidaan sanoa toimivan yhteen ilman liimaa. Ohjelmia on myös helppo laajentaa ja korjata. Koska ohjelmat on jaettu pieniin osiin, jotka eivät vaikuta toisiinsa, uusien ominaisuuksien lisääminen ja vanhan koodin korjaaminen on helppoa.

Seuraavassa esimerkissä luodaan imperatiivisesti if-lauseke, jossa tarkistetaan onko ohjelmalle tuotu parametria sekä asetetaan se ”parameter”-muuttujaan, jos taulukko ei ollut tyhjä:

```
object testProgram {
 def main(args: Array[String])
   var parameter = “”
   if (!args.isEmpty)
     parameter = args(0)
}
```

Scalassa if-lauseke palauttaa arvon toisin kuin esimerkiksi Javassa. If-lauseke voidaan siis asettaa suoraan muuttumattomaan muuttujaan:

```
val parameter =
 if (!args.isEmpty) args(0)
 else “”
```

Nyt ”parameter”-muuttuja ei koskaan muutu. Näin säästytään koodin läpikäymiseltä, jos haluttaisiin varmistaa muuttujan sisältö. Samaa koodia myös voitaisiin käyttää suoraan sellaisenaan muualla. Esimerkiksi:

```
println(parameter)
```

voitaisiin kirjoittaa myös:

```
println(if (!args.isEmpty) args(0) else "")
```

4.5 Selkeys

Funktionaalisesti ohjelmoidun koodin voidaan sanoa olevan selkeämpää ja varsinkin täsmällisempää. Syntaksi voi alkuun tuntua vaikeaselkoiselta, mutta sen ymmärrettyään sillä saadaan aikaiseksi suuri ilmaisuvoima. Korkeamman tason funktiot mahdollistavat rakenteet, jotka vähentävät koodin toistoa. Tämä usein myös selkeyttää samalla koodia. Koska funktio tekee vain yhden asian tehokkaasti, koodin voidaan sanoa olevan lähellä ongelman määrittelyä. Funktiot ovat siten hyvä nimetä selkeästi.

Rekursion voidaan sanoa myös lisäävän selkeyttä. Se on lyhyt ja selkeä toiston esitysmuoto, jolla voidaan moni matemaattinen ongelma ratkaista. Suuret tietomäärät käsitellään usein funktionaalisesti juurikin rekursion avulla. Esimerkiksi for-toistorakenteessa joudutaan miettimään usein aloitetaanko toisto luvusta nolla vai yksi. Rekursiossa tätä ongelmaa ei ole.

4.6 Rinnakkaisuus

Funktionaalisuus mahdollistaa helpon rinnakkaisuuden. Funktioiden suoritus voidaan jakaa useammalle prosessorille, koska funktiot eivät vaikuta toisiinsa. Moniydinprosessorit ovat nykyisin jo hyvin yleisiä ja ne tulevat yleistymään tulevaisuudessa varmasti vielä paljon. Siksi on tärkeätä pystyä jakamaan kuorma tasaisesti kaikille prosessoreille helposti itse ohjelmointikielessä.

Kuorman jakaminen hoidetaan Scalalla actoreiden kautta. Actorit ovat käytännössä samanaikaisia prosesseja, jotka kommunikoivat lähettelemällä toisilleen viestejä. Kuorman jakaminen actoreilla on kuitenkin hyvin laaja aihealue, joten siihen ei tässä opinnäytetyössä enempää keskitytä.

5 POHDINTA

Funktionaalisten ominaisuuksien käyttö Scala-ohjelmointikielessä tehostaa ohjelmointia. Ohjelmakoodista saadaan lyhyempää ja virheettömämpää. Funktionaaliset ominaisuudet kannustavat hajottamaan koodia pienempiin osiin, jotka helpottavat koodin ymmärtämistä. Pienemmät, yhteen ongelmaan keskittyvät, funktiot ovat myös helpompia korjata sekä uudelleenkäyttää, koska funktioilla ei ole sivuvaikutuksia. Funktionaaliset ominaisuudet matemaattisissa ongelmissa, kuten listojen läpikäynnissä, ovat erittäin toimivia.

Tulevaisuudessa ohjelmat tulevat pyörimään moniydinprosessoreissa tai useamalla prosessorilla yhtä aikaa. Tällöin funktionaalisen ohjelmoinnin hyödyt rinnakkaisuuden toteuttamisessa tulevat esille. Funktionaaliset sekä funktionaalisia ominaisuuksia sisältävät ohjelmointikielet tämän myötä mahdollisesti lisäävät suosiotaan. Funktionaalisia ominaisuuksia tullaan myös mahdollisesti lisäämään joihinkin ohjelmointikieliin kuten Javaan ollaan nyt tekemässä.

Syntaksin ja funktionaalisten ominaisuuksien oppiminen vie aikaa. Scalan ollessa myös oliopohjainen, funktionaalisten ominaisuuksien käyttöä voidaan lisätä oman osaamistason mukaan. Osa ongelmista on usein kuitenkin helpompi ratkaista oliopohjaisilla ominaisuuksilla ja Scala rohkaiseekin käyttämään sitä ohjelmointiparadigmaa, joka paremmin soveltuu ongelman ratkaisemiseen.

Syntaktisesti esimerkiksi puhtaiden funktioiden tekeminen ei ole kuitenkaan välttämättä vaikeaa. Pitää vain muistaa, että sivuvaikutuksia ei sallita funktiolle. Funktionaalisessa ohjelmoinnissa onkin tärkeää muistaa ajatella funktionaalisesti.

Funktionaalisesti ohjelmoija oppii tekemään ohjelmia täsmällisemmin. Ohjelmakoodi on myös täten usein luettavampaa ja selkeämpää. Funktionaalisten ominaisuuksien käytön tavoitteena onkin parempia ohjelmia nopeammin.

LÄHTEET

Functional programming. 2011. Viitattu 6.5.2012.
http://www.haskell.org/haskellwiki/Functional_programming

Funktionaalinen ohjelmointi. 1996. Viitattu 9.4.2012.
http://www.mit.jyu.fi/opiskelu/seminaarit/bak/funktion/

Introducing Scala. 2012. Viitattu 25.3.2012.
http://www.scala-lang.org/

Java 8: New Feature. 2012. Viitattu 29.3.2012. http://www.learncomputer.com/java-8-new-features/

List of programming languages in alphabetical order. 2012. Viitattu 22.3.2012.
http://www.scriptol.com/programming/list-programming-languages.php

Odersky, M., Spoon, L. & Venners, B. 2010. Programming in Scala, Second Edition. USA. Artima Press.

Ohjelmointikielen valinta. 2011. Viitattu 25.3.2012.
http://reaktor.fi/osaaminen/ohjelmointikielen_valinta/#footnote1

Ohjelmointikielten kehityshistoriaa. 2006. Viitattu 18.3.2012.
http://www.tol.oulu.fi/kurssit/okp/Luennot/OKP_Historia.html

Peltomäki, J. & Silander, S. 2003. Java 2 Ohjelmoinnin peruskirja. Kustannuspaikka: Docendo Finland Oy.

Tietotekniikan perusteet/1. 2004. Etäopetusmateriaali. Pohjois-Karjalan Aikuisopisto. Viitattu 22.3.2012. http://aikoledu.pkky.fi/ecdl/materiaali/mod1teoria1.pdf

Programming language history. 2012. Viitattu 17.3.2012. http://www.computernostalgia.net/articles/HistoryofProgrammingLanguages.htm

Scala's version fragility make the Enterprise argument near impossible. 2011. Viitattu 25.3.2012. http://lift.la/scalas-version-fragility-make-the-enterprise

The History of Java Technology. n.d. Viitattu 18.3.2012. http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html

TIOBE Programming Community Index for March 2012. Viitattu 18.3.2012. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

Twitter on Scala. 2009. Viitattu 11.5.2012. http://www.artima.com/scalazine/articles/twitter_on_scala.html

Viklo Oy. 2012. Viitattu 18.3.2012. http://www.viklo.fi/